Manual del Laboratorio 2A

2024

ÍNDICE

Manual del Laboratorio 2A

(Elaborado por Johan Baldeón)

1 Introducción

Los Tipos Abstractos de Datos (TAD) son una manera de pensar sobre cómo organizar y manipular datos. Un TAD define un conjunto de operaciones y su comportamiento, pero no especifica cómo se implementa. Este manual explora los TAD más comunes: listas enlazadas, pilas, colas y árboles, y cómo implementarlos en C++/CLI utilizando las bibliotecas de .NET.

2 Conceptos de Tipos Abstractos de Datos (TAD)

2.1 ¿Qué es un TAD?

Un Tipo Abstracto de Datos (TAD) es un modelo matemático para tipos de datos que define el tipo de datos en términos de comportamiento, es decir, qué operaciones se pueden realizar y cómo se comportan esas operaciones. No se preocupa por la implementación de esas operaciones. Los TAD permiten a los desarrolladores concentrarse en la lógica de la aplicación sin preocuparse de cómo están almacenados los datos.

2.2 Importancia de los TAD

- **Modularidad**: Facilita el diseño de software en módulos que son más fáciles de mantener y entender.
- **Reutilización**: Un TAD bien definido puede ser reutilizado en múltiples aplicaciones.
- **Abstracción**: Oculta la implementación específica, lo que permite que los desarrolladores cambien la implementación sin afectar al código que utiliza el TAD.

2.3 Operaciones comunes en TAD

Los TAD generalmente incluyen operaciones básicas, como:

- **Insertar**: Agregar un elemento.
- **Eliminar**: Quitar un elemento.
- **Buscar**: Encontrar un elemento.
- **Acceso**: Obtener un elemento.

3 Listas Enlazadas

3.1 ¿Qué es una lista enlazada?

Una lista enlazada es una estructura de datos lineal compuesta por nodos donde cada nodo contiene un valor y un puntero al siguiente nodo en la lista. Esto permite una inserción y eliminación de elementos eficiente, especialmente en comparación con los arreglos (vectores).

3.2 Tipos de Listas Enlazadas

- **Simplemente Enlazada**: Cada nodo apunta al siguiente nodo.
- **Doblemente Enlazada**: Cada nodo tiene punteros al siguiente y al nodo anterior.
- **Circular**: El último nodo apunta al primer nodo, formando un ciclo.

3.3 Implementación de una Lista Enlazada en C++/CLI

```
#include "stdafx.h"
using namespace System;

// Clase Nodo para la lista enlazada
ref class Node
{
public:
    int data;
    Node^ next;

    Node(int val)
    {
        data = val;
        next = nullptr;
    }
};

// Clase ListaEnlazada para gestionar la lista
ref class LinkedList
{
private:
    Node^ head;

public:
    LinkedList()
    {
        head = nullptr;
    }

    // Método para agregar un nodo al final de la lista
    void Add(int val)
    {
        Node^ newNode = gcnew Node(val);
        if (head == nullptr)
        {
            head = newNode;
        }
        else
        {
            Node^ current = head;
            while (current->next != nullptr)
            {
                current = current->next;
            }
            current->next = newNode;
        }
    }

    // Método para mostrar la lista
    void Display()
    {
        Node^ current = head;
```

```
        while (current != nullptr)
        {
            Console::Write("{0} -> ", current->data);
            current = current->next;
        }
        Console::WriteLine("nullptr");
    }
};

int main(array<System::String ^> ^args)
{
    LinkedList^ list = gcnew LinkedList();
    list->Add(10);
    list->Add(20);
    list->Add(30);
    list->Display(); // Muestra: 10 -> 20 -> 30 -> nullptr

    return 0;
}
```

4 Pilas

4.1 ¿Qué es una pila?

Una pila es una estructura de datos de tipo LIFO (Last In, First Out). Esto significa que el último elemento en entrar es el primero en salir. Las pilas son útiles para aplicaciones como la gestión de memoria, la evaluación de expresiones y la retrocesión de navegación.

4.2 Operaciones comunes en pilas

- **Push**: Agrega un elemento a la parte superior de la pila.
- **Pop**: Elimina y devuelve el elemento de la parte superior de la pila.
- **Peek**: Devuelve el elemento de la parte superior sin eliminarlo.

4.3 Implementación de una Pila en C++/CLI

```
#include "stdafx.h"
using namespace System;
using namespace System::Collections::Generic;

int main(array<System::String ^> ^args)
{
    // Crear una pila de enteros
    Stack<int>^ pila = gcnew Stack<int>();

    // Agregar elementos a la pila
    pila->Push(10);
    pila->Push(20);
    pila->Push(30);

    // Mostrar el elemento superior de la pila sin eliminarlo
    Console::WriteLine("Elemento en la cima: {0}", pila->Peek()); // Muestra: 30

    // Eliminar elementos de la pila
    Console::WriteLine("Elemento eliminado: {0}", pila->Pop()); // Muestra: 30
    Console::WriteLine("Elemento eliminado: {0}", pila->Pop()); // Muestra: 20
```

```
    return 0;
}
```

5 Colas

5.1 ¿Qué es una cola?

Una cola es una estructura de datos de tipo FIFO (First In, First Out). Esto significa que el primer elemento en entrar es el primero en salir. Las colas se utilizan en la gestión de procesos en sistemas operativos, la impresión de documentos, y en la transmisión de datos en redes.

5.2 Operaciones Comunes en Colas

- **Enqueue**: Agrega un elemento al final de la cola.
- **Dequeue**: Elimina y devuelve el primer elemento de la cola.
- **Peek**: Devuelve el primer elemento sin eliminarlo.

5.3 Implementación de una Cola en C++/CLI

```
#include "stdafx.h"
using namespace System;
using namespace System::Collections::Generic;

int main(array<System::String ^> ^args)
{
    // Crear una cola de enteros
    Queue<int>^ cola = gcnew Queue<int>();

    // Agregar elementos a la cola
    cola->Enqueue(10);
    cola->Enqueue(20);
    cola->Enqueue(30);

    // Mostrar el primer elemento de la cola sin eliminarlo
    Console::WriteLine("Elemento al frente: {0}", cola->Peek()); // Muestra: 10

    // Eliminar elementos de la cola
    Console::WriteLine("Elemento eliminado: {0}", cola->Dequeue()); // Muestra: 10
    Console::WriteLine("Elemento eliminado: {0}", cola->Dequeue()); // Muestra: 20

    return 0;
}
```

6 Árboles

6.1 ¿Qué es un árbol?

Un árbol es una estructura de datos jerárquica que consiste en nodos con un valor y enlaces a nodos hijos. Los árboles son útiles en representaciones jerárquicas, búsquedas eficientes, y estructuras de directorios.

6.2 Tipos de Árboles

- **Árbol Binario**: Cada nodo tiene un máximo de dos hijos.
- **Árbol de Búsqueda Binaria (BST):** Un árbol binario donde los nodos a la izquierda son menores que el nodo raíz y los nodos a la derecha son mayores.
- **Árbol AVL**: Un BST auto-balanceado donde las alturas de los subárboles difieren en no más de uno.

6.3 Implementación de un Árbol Binario en C++/CLI

```
#include "stdafx.h"
using namespace System;

// Definición de la clase Nodo del árbol
ref class TreeNode
{
public:
    int value;
    TreeNode^ left;
    TreeNode^ right;

    TreeNode(int val)
    {
        value = val;
        left = nullptr;
        right = nullptr;
    }
};

// Definición de la clase Árbol Binario
ref class BinaryTree
{
public:
    TreeNode^ root;

    BinaryTree()
    {
        root = nullptr;
    }

    void Insert(int val)
    {
        root = InsertRec(root, val);
    }

private:
    TreeNode^ InsertRec(TreeNode^ node, int val)
    {
        if (node == nullptr)
        {
            return gcnew TreeNode(val);
        }

        if (val < node->value)
        {
            node->left = InsertRec(node->left, val);
        }
        else if (val > node->value)
        {
            node->right = InsertRec(node->right, val);
```

```
        }

        return node;
    }

public:
    void InOrder()
    {
        InOrderRec(root);
    }

private:
    void InOrderRec(TreeNode^ node)
    {
        if (node != nullptr)
        {
            InOrderRec(node->left);
            Console::Write("{0} ", node->value);
            InOrderRec(node->right);
        }
    }
};

int main(array<System::String ^> ^args)
{
    BinaryTree^ tree = gcnew BinaryTree();
    tree->Insert(50);
    tree->Insert(30);
    tree->Insert(20);
    tree->Insert(40);
    tree->Insert(70);
    tree->Insert(60);
    tree->Insert(80);

    Console::WriteLine("Recorrido en orden del árbol:");
    tree->InOrder(); // Muestra: 20 30 40 50 60 70 80

    return 0;
}
```